Definite Loops
Methods
If-else statements
Parameter Passing
Scope
Calling stack

Definite Loops

Many problem solutions involve the repetition of a task for a fixed number of times
- Calculate average by adding numbers one after the other to a sum, then divide by the count of numbers
- Print a list of items
- Perform a simulation for a number of times
- Use the for loop in Java for a fixed number of repetitions
  - Java also has general (indefinite) looping
  - for loop can be used for indefinite looping, but we will start with fixed numbers of repetitions
For Loop

- Use a loop to execute a block of statements a fixed number of times
  - Initialize some "counting" variable
  - Check that the counter is within the range
  - Execute the statements
  - Increment the counter
  - If the counter is within the range, repeat
- The for loop construct makes this pattern easy to code
  - Could use other loop forms, but a for loop makes the logic for fixed repetitions much more obvious
  - A for loop helps distinguish between the mechanics of the loop and the work that is to be done repeatedly

For Loop (Definite form)

- The definite for loop has four parts
  - Initialization of the counting variable
  - Limit comparison for the counting variable
  - Body which is executed if the limit is not reached
  - Increment of the counting variable
- After body and iterator are executed, limit is checked again
- Syntax of for statement
  ```
  for (initialize; check limit; increment) {
    statements;
  }
  ```
  - Initialization done once
  - Parentheses required
  - Semicolons required
  - Increment after the statements
  - Check if limit is reached
  - Code to execute for each repetition
Flow chart of for loop

```
howmany = 3; start = 5;
number = 7; sum =18;  count = 4;

for (int count = 1;
     count <= howmany;
     ++count) {
    sum += ++number;
}
System.out.println("Sum from " + start + " to " + number + " is " + sum);
```

RangeSum2.java

Methods

- Why methods?
- Coding a method
- Calling method
- Method parameters
Procedural Decomposition

- Controlling complexity is a key concept in problem solving and software design
- We deal with complexity by breaking complex tasks into smaller, more manageable tasks
  - And smaller tasks may be divided into yet smaller tasks...
- For example, a recipe for cookies may be decomposed into
  - Preparation (preheat oven, grease pan)
  - Make the dough (measure, mix ingredients, blend)
  - Bake (Measure dough, bake, test, cool)
- The recipe is still ultimately a sequence of individual steps, but the higher level grouping allows us to more readily understand the overall process
  - Procedural decomposition focuses on the steps or actions
  - Object oriented decomposition focuses on the entities (later)

Building Blocks of Programs

- A program consists of many statements
- It is possible to code any logic using just if-else and loops
  - But for a large program, this would be tedious
  - There is likely to be a lot of duplication of logic
- E.g., computing the square root of a number takes a lot of steps
  - If a program needed to compute many square roots, duplicating all these steps would be error prone, hard to read and understand
- Better solution would be to have a “black box” to contain the square root logic
  - Then we could use the black box many times
  - ...and only look inside when we need to fix it or understand how it works
- Methods are the black boxes of programming
  - Similar to formulas in math
Java Methods

- A sequence of statements can be grouped together as a **method** (sometimes called a function or procedure)
  - We have already been doing that with **main**
  - We can define methods other than main
- Methods that we define may be **called**
  - Changes flow of control – jump to method, execute statements, return to point of call
  - Methods may be called repeatedly (**re-entrant** code)
- Definition of method
  - A method has a **name**
  - A method has a **return type**
  - A method has zero or more **parameters**
  - A method has a **body** (the statements)

Coding a Method Example

- Simulate coin tosses, ten at a time, until user decides to quit
  - Prompt to start
  - Prompt to continue after each ten tosses

  **Toss.java**

- Code to prompt before loop and inside loop can be factored out into a method named **prompt**

  - Note that loop is terminated by program exiting from the prompt method
Reusing Code

```java
System.out.println("This is a simulation of flipping coins...");
System.out.println("--------------");
System.out.print("Press return to continue or CTRL-C to quit. ");
try { scan.nextLine(); } 
catch (Exception e) {
    System.out.println();
    System.out.println("Simulation finished");
    System.exit(0);
}
for (int i = 0; i < 100; ++i) {
    for (int k = 0; k < 10; ++k) {
        ++count;
        won += (int)(Math.random() * 2);
    }
    System.out.printf("After %d tosses, you have . . ");
    System.out.println("--------------");
    System.out.print("Press return to continue or CTRL-C to quit ");
    try { scan.nextLine(); } 
    catch (Exception e) {
        System.out.println();
        System.out.println("Simulation finished");
        System.exit(0);
    }
}
```

Method Definition and Call

- Static method **definition** with no return value and no parameters

```
public static void prompt() {
    // Code of the method
}
```

- Statement to **call** the method

```
prompt();
```
Simplify with method call

- **Define** `prompt` method

```java
public static void prompt() {
    Scanner scan = new Scanner(System.in);
    System.out.println("------------------");
    System.out.print("Press return to continue or CTRL-C to quit ");
    try {
        scan.nextLine();
    } catch (Exception e) {
        System.out.println();
        System.out.println("Simulation finished");
        System.exit(0);
    }
}
```

- **Replace prompting code blocks with method call**

```java
for (int i = 0; i < 100; ++i) {
prompt();
System.exit(0);
System.out.println("Simulation finished");
System.out.println();
}
```

- **Benefits of Methods**

  - A method is an **abstraction**
    - Emphasis on operation
    - Hides the details of the implementation
  
  - Methods facilitate the **reuse** of code
    - Don't have to re-design the implementation
    - Makes software maintenance easier
    - Change the method implementation, and all calls will use the changes

Static Methods

- A class may contain one or more static methods
  - `main` is a static method when that class is the starting point for the whole program
- Static methods may be called from any other method
  - If called from a method in the same class, we only need the method name, e.g., `prompt()`
  - If called from a method in a different class, then the method name must be qualified by the class name, e.g., `Toss2.prompt()`
  - No objects need to be created in order to call static methods
  - The `Math` class contains many useful static methods
- In other languages, static methods would be called procedures or functions

Using Java Math methods

- All Math methods are static
- No import needed since class name is used, e.g.,
  ```java
double val = Math.random();
```
  - Could use static import:
    ```java
    import static java.lang.Math.*;
    ...
double val = random();
    ```
- There are no Math objects, just static methods
  - Math functions: abs, max, min, sqrt, pow, exp, log
  - Trig functions: sin, cos, tan
  - Rounding functions: ceil, floor, round
  - Useful constants: PI, E
### Java API Documentation

- Java classes are documented in the Application Programming Interface (API)
- Web pages, one page for each class
  - Over 7,000 pages!
  - Well indexed, with links between related classes
- For each class, API gives
  - Description of class, inheritance hierarchy
  - List of constructors – arguments, synopsis
  - List of all public methods – arguments, return value, synopsis
  - Sometimes example code
  - Math API page

---

### Boolean Values

- Recall `boolean` data type
  - May only have the values `true` or `false`
  - `true` and `false` are constants, just like numerical constants
- Comparison operators result in a boolean value
  - “Equal to” is the operator: `==`
  - “Not equal to” is the operator: `!=`
  - Less than: `<` Less than or equal to: `<=`
  - Greater than: `>` Greater than or equal to: `>=`
- Example
  ```java
  int n, m;
  
  boolean same = (n == m);
  boolean different = (n != m);
  ```
Boolean Operators

- Boolean values can be combined using logic operators to form other boolean values
- Logical negation (not) operator  
  - Applies to a single boolean value
  - Value of operation is true if operand is false
  - Value of operation is false if operand is true
  
    ```java
    boolean opposite = ! answer;
    ```
- Logical conjunction (and) operator &&
  - Applies to two boolean operands
  - Value is true only if both operands are true, otherwise false
  
    ```java
    boolean both = test1passed && test2passed;
    ```
- Logical disjunction (or) operator ||
  - Applies to two boolean operands
  - Value is false only if both operands are false, otherwise true
  
    ```java
    boolean atLeastOne = test1passed || test2passed;
    ```
- Logical exclusion (exclusive or) operator ^
  - Applies to two boolean operands
  - Value is true only if exactly one operand is true, otherwise false
  
    ```java
    boolean justOne = test1passed ^ test2passed;
    ```

Truth table for boolean operators

| a  | b  | !a | a&&b | a||b | a^b |
|----|----|----|------|------|-----|
| true| true| false| true | true | false |
| true| false|       | false | true | true |
| false| true| true|      | true | true |
| false| false|       | false | false | false |
### Boolean Expressions

- True if a number is even and positive
  - \((num \% 2 == 0) \&\& (num > 0)\)

- True if the three numbers are in ascending order
  - \((num1 <= num2) \&\& (num2 <= num3)\)

- True if either two or three divide a number, but it cannot be zero
  - \((\neg(\neg(num \% 2 == 0) || (\neg(num \% 3 == 0)) \&\& (num != 0)\))\)

- True if neither three nor five divide a number
  - \(!((num \% 3 == 0) || (num \% 5 == 0))\)

- True if only one number divides the other
  - \((num1 \% num2 == 0) ^ (num2 \% num1 == 0)\)

### Conditional Operator

- The conditional operator selects one of two expressions to evaluate
  - The value of the conditional operator is the selected expression value
  - Only one of the expressions is evaluated
  - Operator consists of two symbols, ? and : - to separate 3 operands
  - This is the only operator in Java that takes 3 operands
  - Like if-else flow control, but selects expression, not a statement

- Example
  - Set the variable \textbf{max} to be the larger of two variables \textbf{a} and \textbf{b}

  \[
  \text{max} = (a > b) \ ? \ a : b ;
  \]
**if Statements**

- The code of a program is a sequence of statements, executed in order.
- The `if` control flow construct allows a statement or block of statements to be executed or not, according to a condition.

**Syntax of if statement**

```
if (condition) {
  statements;
}
```

- `keyword if`
- `a boolean expression`
- `parentheses required`
- `code to execute only if condition is true`
- `braces required if more than one statement`

---

**Flow chart of if statement**

```
n = 17; m = 3;

if (n % m == 0) {
  System.out.println(m + " divides " + n);
}
```

```
Divides.java
```

Week 2  
CIS 210  
Winter 2011  
23

Week 2  
CIS 210  
Winter 2011  
24
if-else Statements

- Sometimes we want to select between two choices of code to execute.
- The if-else control flow construct selects between two statements (or blocks) executing one or the other (but not both), according to a condition.

Syntax of if-else statement

```
if (condition) {
    statements;
} else {
    statements;
}
```

code to execute only if condition is true

code to execute only if condition is false

Flow chart of if-else statement

```
n = 17; m = 3;

if (m > n) {
    biggest = m;
} else {
    biggest = n;
}
```

Bigger.java
Nested Statements

- A block of statements can appear in any place where you could code a single statement
- An if construct and an if-else construct are statements themselves
  - E.g., an if can control an if-else statement
- Braces may be important to get the correct control logic
  - Each else must be matched with an if
  - An else matches to the "closest" if (respecting braces)
- When the else clause is an if or if-else, we use the "else if" style
  ```java
  if (m > n)
      System.out.println(m + " is bigger");
  else if (n > m)
      System.out.println(n + " is bigger");
  else
      System.out.println("they are the same");
  ```

More about Methods

- Methods can be more versatile when they are parameterized and return values
- Method Parameters
  - Parameter values are passed to methods
  - Code of the method is like a formula that is applied to the parameter values
  - Method definition must declare each parameter type
  - Method call must provide values that match the formal parameters of the method definition
- Return values
  - A method can return a single value
  - The type of the value returned must be given in the method definition
  - The value is returned by using the `return` statement
Method to Compute Maximum

- Static method **definition** with two integer parameters and an integer return value

```java
public static int max(int num1, int num2) {
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```

- **MaxTest.java**

Method Control Flow

```
x  5
y  13
m 13
```

```java
void main(...) {
    ...  
    int x = 5;
    int y = 13;
    int m = max(x, y);
    System.out.println(m);
}
```
Variable Scope

- The **scope** of a variable is the places in the Java source code where it is legal to use the variable.
  - Scope also applies to all identifiers: method names, parameter names, class names.
- The scope of a local variable is all the lines from the line of declaration up to the end of the enclosing block.
  - For a local variable, you can think of the variable as coming into existence at the point of declaration and disappearing at the end of the block.
  - A variable declared in a for loop has scope restricted to the body of the loop.
- Variables may be declared in any block.
  - The same names may be used for variables in non-overlapping blocks.
- Parameters to a method are like local variables.
  - The point of declaration is the method header.
  - The scope extends to the end of the method body.

```java
int power(int x, int y) {
    int result = 1;
    for (int i = 0; i < y; ++i) {
        int tmp = result;
        for (int j = 1; j < x; ++j)
            result = result + tmp;
    }
    return result;
}

void main(String[] args) {
    int x = 2;
    int y = 5;
    for (int i = 0; i <= y; ++i) {
        System.out.println(x + " to the " + i + " power is " + power(x,i));
    }
}
```
Method Stacks

- Each call to a method is like placing an order and waiting for it to be filled
  - To execute the code of the method, an environment is needed for the method parameters, the local variables of the method, and the return value
- The code of one method may call another method
  - The first method "waits" for the second method it calls to finish
- The order must be preserved
  - The order is last in, first out (the last method called must finish and return before the method that called it can continue)
- Method calls form a stack
  - The first method is on the bottom
  - The last method called is on the top
  - When the last method called finishes, it is popped from the stack
  - The environment for each call is called a stack frame

Tracing the Method Stack

```java
int max(int n1, int n2) {
    if (n1 > n2)
        return n1;
    else
        return n2;
}

int max3(int n1, int n2, int n3) {
    return max(max(n1, n2), n3);
}

void main() {
    m = max3(x, y, z);
}
```

```
x 89
y 117
z 91
m 117
```
Overloading Methods

- Different methods should have different names
- But methods that implement similar abstractions and only differ by their parameters may have the same name
  - This is called **method overloading**
  - The **number of parameters** may be different
  - Or, the **types of the parameters** may be different
  - Java figures out which method to call based on the number and types of the actual parameters used
- The method name, along with the number of parameters and their types, is called the **method signature**
  - It uniquely identifies the method

MaxOverload.java